XCS with Stack-Based Genetic Programming

Pier Luca Lanzi
Artificial Intelligence and Robotics Laboratory
Dipartimento di Elettronica e Informazione
Politecnico di Milano
lanzi@elet.polimi.it
Voice +39-02-23993472
Fax +39-02-23993411

Abstract- We present an extension of the learning classifier system XCS in which classifier conditions are represented by RPN expressions and stack-based Genetic Programming is used to recombine and mutate classifiers. In contrast with other extensions of XCS involving tree-based Genetic Programming, the representation we apply here produces conditions that are linear programs, interpreted by a virtual stack machine (similar to a pushdown automaton), and recombined through standard genetic operators. We test the version of XCS extended with stack-based conditions on a set of problems of different complexity.

1 Introduction

Learning classifier systems (LCSs) are rule-based systems which exploit *reinforcement learning* [19] and *genetic algorithms* [6] to extract interesting rules (the classifiers) from a set of examples [16]. In their original description, learning classifier systems assume that inputs and outputs are coded by binary strings and, *most important*, that classifier conditions are strings on the ternary alphabet {0,1,#}. The symbol #, called don't care, means that the corresponding position can match either a 0 either a 1. Note however that this assumption was intended as a simplification of the framework rather than an actual limitation (e.g., [7]). Along the years, there have been many proposals for enhancing the representation capabilities of learning classifier systems so as to allow the representation of high level knowledge. Although such proposals date back to the very early years of learning classifier system research (e.g., [18]) only recently implementations of advanced representations have been presented (with probably the only exception of [20]). The vast majority of these extensions have been implemented on Wilson's XCS [21] which nowadays can be considered one of the best performing model of learning classifier systems. In particular, [12] borrowed the work on messy genetic algorithms and extended XCS with *messy conditions*; Wilson introduced conditions based on integer and real intervals in [22, 23]; Bull (e.g., [4]) proposed classifiers with conditions represented as neural networks and also with actions represented by means of Genetic Programming [1]. Finally, in [15, 13] we used tree-based Genetic Programming to represent classifier conditions and applied XCS with Lisp-like s-expressions to some supervised classification problem. One of the major computational costs of applying tree-based Genetic Programming, apart from the matching of rule conditions, is caused by the genetic operators, crossover and mutation, which in their most common form must examine the large part of parent conditions before offspring can be generated.

To reduce the computational cost involved by the genetic operators borrowed from tree-based Genetic Programming, in this paper, we extend the learning classifier system XCS with conditions based on stack-based Genetic Programming as introduced in [17]. Classifier conditions are now linear programs written using Reverse Polish Notation (RPN) and they are interpreted by a virtual stack machine (similar to a pushdown automaton). Recombination and mutation are implemented using standard operators borrowed from Genetic Algorithms, and therefore, in contrast with the operators from linear encoded Genetic Programming, we are not guaranteed that offspring conditions will be syntactically correct. In contrast, we note that all the representations tested so far with learning classifier systems used genetic operators that generated syntactically corrected offspring. We test the version of XCS extended with stack based conditions on both single step problems involving the learning of Boolean functions and multistep problems involving small grid worlds, called *woods* environment. The results we present show that XCS with stack-based Genetic-Programming can perform optimally in all the problems presented here. All the experiments reported here have been carried out with `xcslib` [14]; the files needed to replicate the results are available on request.

The remainder of this paper is organized as follows. In Section 2 we overview the XCS classifier system as introduction in [21] while in Section 3 we discuss how stack-based representation can be added to XCS. In Section 4 we present the design of experiments. In Section 5 we apply XCS with stack-based representation to the 6-multiplexer and to the 11-multiplexer, and in Section 6 we apply it to the `Woods1`. In Section 7 we draw some conclusions and highlight possible future research directions.

2 The XCS Classifier System

Classifiers in XCS consist of a condition, an action, and four main parameters: (i) the prediction p, which estimates the payoff expected when the classifier is used; (ii) the prediction error ϵ, which estimates the error of the prediction p; (iii) the fitness F, which estimates the *relative* accuracy of the payoff prediction given by p; and finally (iv) the numerosity *num*, which indicates how many copies of classifiers with the same condition and the same action are present in the population. Note that in the population [P] no duplicates classifiers exist, i.e., there is *only one* classifier with a

0-7803-7804-0/03/$17.00 © 2003 IEEE

certain condition-action pair.[1]

Performance Component. At each time step, XCS builds a *match set* [M] containing the classifiers in the population [P] whose condition matches the current sensory inputs; if [M] contains less than θ_{nma} actions, *covering* takes place and creates a new classifier that matches the current inputs and has a random action. For each possible action a_i in [M], XCS computes the *system prediction* $P(a_i)$ which estimates the payoff that the XCS expects if action a_i is performed. The *system prediction* is computed as as the fitness weighted average of the predictions of classifiers in [M], $cl \in$[M], which advocate action a_i (i.e., $cl.a$=a_i):

$$P(a_i) = \frac{\sum_{cl.a=a_i \wedge cl \in [M]} cl.p \times cl.F}{\sum_{cl.a=a_i \wedge cl \in [M]} cl.F} \quad (1)$$

where, following the notation of [5], $cl.a$ is the action of classifier cl, $cl.p$ is the prediction of classifier cl, and $cl.F$ is the fitness of classifier cl. Then XCS selects an action to perform; the classifiers in [M] which advocate the selected action form the current *action set* [A]. The selected action is performed in the environment, and a scalar reward R is returned to XCS together with a new input configuration.

Reinforcement Component. When the reward R is received, the parameters of classifiers in [A] are updated in the following order [5]: prediction, prediction error, and finally fitness. Prediction p is updated with learning rate β ($0 \leq \beta \leq 1$): $p \leftarrow p + \beta(R - p)$. Similarly, the prediction error ϵ is updated as: $\epsilon \leftarrow \epsilon + \beta(|R - p| - \epsilon)$

Fitness Update. The update of classifier fitness consists of three steps. First, the *raw accuracy*[2] κ of the classifiers in [A] is computed as:

$$\kappa = \begin{cases} 1 & \text{if } \epsilon \leq \epsilon_0 \\ \alpha(\epsilon/\epsilon_0)^{-\nu} & \text{otherwise} \end{cases} \quad (2)$$

A classifier is *accurate* if its prediction error ϵ is smaller than the threshold ϵ_0 so that its *raw accuracy* κ is one. A classifier is *inaccurate* if its prediction error ϵ is larger than ϵ_0; the *raw accuracy* κ of an inaccurate classifier is computed as a potential descending slope given by $\alpha(\epsilon/\epsilon_0)^{-\nu}$. The parameter ϵ_0 ($\epsilon_0 > 0$) is the threshold that determines to what extent prediction errors are accepted; α ($0 < \alpha < 1$) causes a strong distinction between accurate and inaccurate classifiers; ν ($nu > 0$), together with ϵ_0, determines the steepness of the slope used to calculate classifier accuracy. The *raw accuracy* κ is used to calculate the *relative accuracy* κ' as:

$$\kappa' = \frac{(\kappa \times num)}{\sum_{cl \in [A]} (cl.\kappa \times cl.num)} \quad (3)$$

where $cl.\kappa$ is the *raw accuracy* of classifier cl, as computed in equation 2; $cl.num$ is the *numerosity* of classifier cl. Finally the *relative accuracy* κ' is used to update the classifier fitness as: $F \leftarrow F + \beta(\kappa' - F)$.

Discovery Component. On regular basis, roughly every θ_{ga} steps, the genetic algorithm is applied to classifiers in [A]. It selects two classifiers with probability *proportional to their fitnesses*, copies them, and with probability χ performs crossover on the copies; then, with probability μ it mutates each allele. The resulting offspring classifiers are inserted into the population and two classifiers are deleted to keep the population size constant.

3 Adding Stack Based Representation

Stack-based Genetic Programming as introduced by [17] is a simplification of the first implementation presented in [10] (see also [3, 8]). In fact, [17] uses the same representation (i.e., linear program represented in Reverse Polish Notation, or RPN), and the same approach to computation (i.e., a pushdown automaton), but, employs standard *genetic* operators for crossover and mutation, instead of the Genetic Programming operators developed for linear encoding, such for instance those discussed in [8]. Accordingly, [17] does not guarantee that offspring are syntactically correct.

It is quite straightforward to extend XCS with stack-based Genetic Programming. As we have done in [15], we must define (i) the syntax of classifier conditions, i.e., the terminal symbols, the function symbols, and the constants involved, (ii) how classifier conditions are matched against the incoming sensory inputs, (iii) how covering conditions are generated when an unmatched input configuration is presented, and finally, (iv) how genetic operators work.

Conditions. Classifier conditions are linear programs expressed in Reverse Polish Notation. Conditions are sequences of tokens; each token can be either a variable, a constants, or a function. Variables represent the values of sensory inputs, accordingly for every input j a variable X_j is defined. Constants are numeric values that usually belongs to the space of the sensory input values; note that when applying Genetic Programming to classifier systems ephemeral constants are not used but actual constant values are employed. The set of functions we use in this work includes Boolean operators (AND, OR, NOT, EOR), arithmetic functions (+ and -), and comparisons (> and =).

Matching. This involves the execution of the classifier conditions against the current sensory inputs. For this purpose a pushdown automaton is used. The condition is passed from left to right: if the current token is a constant, the corresponding value is pushed onto the stack; if the current token is a variable, the input value of the corresponding sensor is pushed onto the stack; if the current token is a function *and* there are enough values in the stack, the arguments are popped out from the stack, the function is computed, and the result pushed back onto the stack; otherwise,

[1] In contrast with Wilson [21], in this paper we do not distinguish between classifiers and macroclassifiers. For the sake of simplicity, we prefer the above definition that includes both the ideas all together.

[2] Note that we prefer the term *raw accuracy* rather than the more intuitive *absolute accuracy*, to put in more evidence that κ (the *raw accuracy*) is an *estimate* of what is the true (i.e., absolute) accuracy of the classifier.

if there are not enough values on the stack to compute the function, the function is ignored.

Covering. The covering operators works basically as in XCS and it is controlled by the same parameter, i.e., the don't care probability $P_{\#}$. When no classifier condition matches the current sensory input, a covering condition is created as follows. With probability $P_{\#}$, XCS decides which sensory inputs will be covered; for each of these an elementary expression matching the corresponding input is added to the condition, note that variables are used to represent current input values; a sufficient number of Boolean *and* are added to the condition in order to build a logical disjunction.

Genetic Operators. Crossover and mutations work basically as in XCS. Crossover is activated with probability χ, it selects two parent classifiers, copies them, and applies a single point crossover to the classifier conditions (note that since the discovery component acts in [A], classifiers have the same action). Then with probability μ it changes the value of every token in the conditions of offspring classifiers.

4 Experimental Design

The experiments reported in this paper were performed following the standard settings used in the literature [21]. Each experiment consists of a number of problems that the system must solve. When the system solves the problem correctly, it receives a constant reward equal to 1000; otherwise it always receives a constant reward equal to 0. Each problem is either a *learning* problem or a *test* problem. In *learning* problems, the system selects actions randomly from those represented in the match set. In *test* problems, XCS always selects the action with the highest prediction. The genetic algorithm is enabled only during *learning* problems, while it is turned off during *test* problems. The covering operator is always enabled, but operates only if needed. Learning problems and test problems alternate. After the learning has stopped, an additional *condensation* phase is activated [13]. During condensation, the genetic algorithm is functioning but crossover and mutation are turned off; condensation causes the population to shrink dramatically due to increase of the selective pressure towards high fitness classifiers while inhibiting the creation of new offspring. The performance is computed as the moving average of the correctly classified examples in the last 100 test problems. All the statistics reported in this paper are averaged over 10 experiments.

Boolean Multiplexer. These are defined for l Boolean variables (i.e., bits) where $l = k + 2^k$: the first k variables $(x_0 \ldots x_{k-1})$ represent an address which indexes into the remaining 2^k variables $(y_0 \ldots y_{2^k-1})$; the function returns the value of the indexed variable. For example, consider the multiplexer with 20 variables

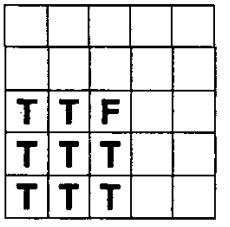

Figure 1: The `Woods1` environment.

$\mathrm{mp}_{20}(x_0, x_1, x_2, x_3, y_0, \ldots, y_{15})$ is defined as follows:

$$\begin{aligned}\mathrm{mp}_{20}(x_0, x_1, x_2, x_3, y_0 \ldots, y_{15}) = \\ &\overline{x_0}\,\overline{x_1}\,\overline{x_2}\,\overline{x_3}\,y_0 + \overline{x_0}\,\overline{x_1}\,\overline{x_2}\,x_3\,y_1 + \overline{x_0}\,\overline{x_1}\,x_2\,\overline{x_3}\,y_2 + \\ &\overline{x_0}\,\overline{x_1}\,x_2\,x_3\,y_3 + \overline{x_0}\,x_1\,\overline{x_2}\,\overline{x_3}\,y_4 + \overline{x_0}\,x_1\,\overline{x_2}\,x_3\,y_5 + \\ &\overline{x_0}\,x_1\,x_2\,\overline{x_3}\,y_6 + \overline{x_0}\,x_1\,x_2\,x_3\,y_7 + x_0\,\overline{x_1}\,\overline{x_2}\,\overline{x_3}\,y_8 + \\ &x_0\,\overline{x_1}\,\overline{x_2}\,x_3\,y_9 + x_0\,\overline{x_1}\,x_2\,\overline{x_3}\,y_{10} + x_0\,\overline{x_1}\,x_2\,x_3\,y_{11} + \\ &x_0\,x_1\,\overline{x_2}\,\overline{x_3}\,y_{12} + x_0\,x_1\,\overline{x_2}\,x_3\,y_{13} + x_0\,x_1\,x_2\,\overline{x_3}\,y_{14} + \\ &x_0\,x_1\,x_2\,x_3\,y_{15}\end{aligned}$$

The product corresponds to logical *and*, the sum to logical *or*, and the overline corresponds to logical not. The system goal is to learn how to represent the Boolean multiplexer from a set of examples. For each problem, the system receives as input an assignment of the input variables; the system has to answer with the corresponding truth value of the function (0 or 1); if the answer is correct the system is rewarded with 1000, otherwise 0.

Woods Environments. These are simple grid worlds like those depicted in Figure 1, which contains obstacles (`T`), free positions (`.`), and food (`F`). There are eight sensors, one for each possible adjacent cell. Each sensor is encoded with two bits: `10` indicates the presence of an obstacle `T`; `11` indicates a goal `F`; `00` indicates an empty cell. Classifiers conditions are 16 bits long (2 bits $\times$ 8 cells). There are eight possible actions, encoded with three bits. The system goal is to learn how to reach food position from any free position; the system is always in one free position and it can move in any surrounding free position; when the system reaches a food position (`F`) the problem ends and the systems is rewarded with 1000; in all the other cases the system receives zero reward.

5 Experiments with Boolean Multiplexer

In the first experiment, we apply XCS with stack-based conditions to the 6-multiplexer when the population size N is 400 classifiers, $P_{\#} = 0.3$, β=0.2, χ=0.8, μ=0.04, θ_{nma}=2, ϵ_0=10, χ=0.8, μ=0.04, θ_{nma}=2, and θ_{del}=20. Condensation starts after 50000 learning problems and lasts for 50000 problems.

Figure 2 reports the performance of XCS computed as the percentage of correctly classified examples (solid line) and the percentage of classifiers in the population (dashed line). XCS reaches optimal stable performance quite rapidly, by

10000 learning problems. As reported with usual tree-based Genetic Programming in [15, 13], also with stack-based representation there is an immediate bloat of the population [9, 2, 11]: the number of macroclassifiers in the population almost immediately reaches the 80% and remains stable around 85%. Then, when condensation is activated the number of macroclassifiers dramatically drops and after 50000 problems with condensation activated (i.e., the GA is activated but crossover and mutation are turned off) the percentage of classifiers in the population size is around the 4% of the population size N, i.e, 16 classifiers.

We extend the previous results and we apply XCS with stack-based to the 11-multiplexer when the population size N is 1000 classifiers, the other parameters are set as in the previous experiment. condensation starts after 100000 learning problems and lasts for 50000 problems. Figure 3 reports the performance of XCS computed as the percentage of correctly classified examples (solid line) and the percentage of classifiers in the population (dashed line). XCS reaches optimal stable performance around 60000 learning problems although it is already very near to the optimum by 30000 problems. As in the experiment with the 6-multiplexer, the number of macroclassifiers in the population almost immediately reaches the 80% and remains stable around 85%; when condensation is activated after 100000 problems, the number of macroclassifiers dramatically drops and after 50000 problems the average population size is around the 3%, i.e., more or less 30 classifiers.

6 Experiments with Woods1

We now apply XCS with stack-based Genetic Programming to the simple multistep environment named `Woods1` (Figure ??). Population size is 1000 classifiers, the discount factor γ is 0.7, the probability $P_{\#}$ is 0.3, while all the other parameters are set as in the previous experiments. Condensation starts after 10000 learning problems and last for 5000 problems.

Figure 4 reports the performance of XCS computed as the average number of steps to the goal position during the last 100 problems (Figure 4a) and the percentage of classifiers in the population (Figure 4b). Figure 4a shows that XCS reaches optimal performance, represented by the horizontal line at 1.68; likewise to the experiments with the Boolean multiplexer, population tends to bloat immediately reaching more or less the 90% stably until condensation starts after 10000 problems; when condensation starts the number of macroclassifiers in the population drops dramatically reaching over ten experiments an average of 30 classifiers. Note that, as in all the other experiments, the performance remains optimal although the population is shrinking due to condensation.

7 Summary

We extended XCS by adding a representation of classifier conditions that is borrowed from stack-based Genetic Programming. We tested the new version of XCS on different

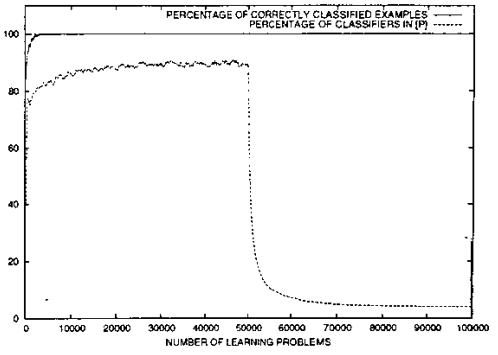

Figure 2: XCS with stack-based representation in the 6-multiplexer. Percentage of correctly classified examples (solid line). Percentage of classifiers in the population (dashed line). Population size $N = 400$. Curves are averages over 10 experiments.

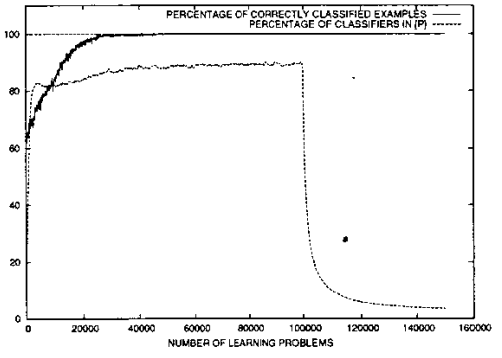

Figure 3: XCS with stack-based representation in the 11-multiplexer. Percentage of correctly classified examples (solid line). Percentage of classifiers in the population (dashed line). Population size $N = 400$. Curves are averages over 10 experiments.

problems showing that XCS always performed optimally. This result is quite interesting. This representation easily generates conditions that are not syntactically correct since genetic operators do not take into account any information about operators structure and arity. Accordingly, the search space of the feasible solutions is highly redundant, even more than with symbolic conditions (as those we used in [15]). Nevertheless, XCS can still learn optimal behavior even with small population sizes. In fact, if we compare the values of N used in the experiments presented here, we note that they are very near to those used for the original version of XCS, based on the simpler ternary representation. Future work includes a comparison between ternary, symbolic, and stack-based representation, regarding the performance in terms of computation speed; as well as the application of stack-based representation on more difficult problems.

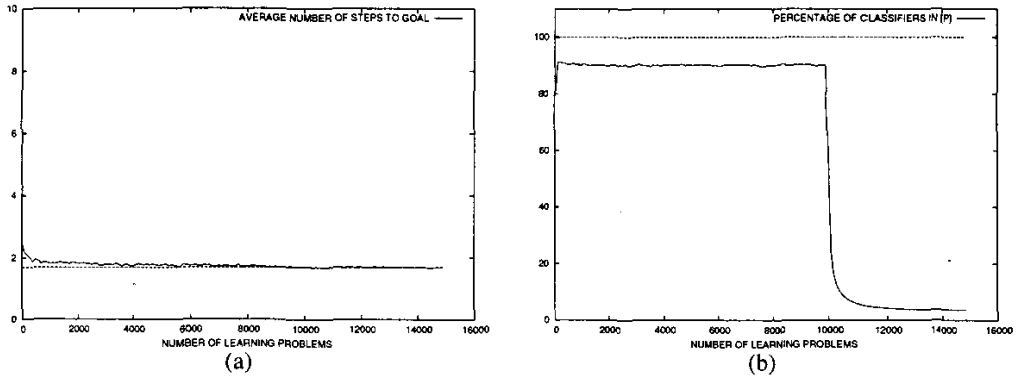

Figure 4: XCS with stack-based representation in in `Woods1`. (a) The performance computed as the average number of steps to the goal position; (b) The percentage of classifiers in the population. Curves are averages over 10 experiments.

Bibliography

[1] Manu Ahluwalia and Larry Bull. A Genetic Programming-based Classifier System. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99)*, pages 11–18. Morgan Kaufmann: San Francisco, CA, 1999.

[2] Peter J. Angeline. Subtree crossover causes bloat. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 745–752, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[3] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag, January 1998.

[4] Larry Bull and Matt Studley. Consideration of multiple objectives in neural learning classifier systems. In J.-J. Merelo Guervós, P. Adamidis, H.-G. Beyer, J.-L. Fernández-Villaca nas, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VII, 7th International Conference, Granada, Spain, September 7-11, 2002. Proceedings*, number 2439 in Lecture Notes in Computer Science, LNCS, page 549 ff. Springer-Verlag, 2002. Keywords: Related::Evolutionnary Robotics, Technique::Classifier systems, Technique::Multi-objective.

[5] Martin V. Butz and Stewart W. Wilson. An algorithmic description of xcs. *Soft Computing – A Fusion of Foundations, Methodologies and Applications*, 6(3–4):144–153, 2002.

[6] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975. Republished by the MIT press, 1992.

[7] John H. Holland. Escaping Brittleness: The possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems. In Mitchell, Michalski, and Carbonell, editors, *Machine learning, an artificial intelligence approach. Volume II*, chapter 20, pages 593–623. Morgan Kaufmann, 1986.

[8] Mike J. Keith and Martin C. Martin. Genetic programming in C++: Implementation issues. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 13, pages 285–310. MIT Press, 1994.

[9] John Koza. *Genetic Programming*. MIT Press, 1992.

[10] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[11] William B. Langdon, Terry Soule, Riccardo Poli, and James A. Foster. The evolution of size and shape. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA, June 1999.

[12] Pier Luca Lanzi. Extending the Representation of Classifier Conditions Part I: From Binary to Messy Coding. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 99)*, pages 337–344, Orlando (FL), July 1999. Morgan Kaufmann.

[13] Pier Luca Lanzi. Mining interesting knowledge from data with the xcs classifier system. In Lee Spector, Erik D. Goodman, Annie Wu, W.B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic*

and Evolutionary Computation Conference (GECCO-2001), pages 958–965, San Francisco, CA 94104, USA, 7-11 July 2001. Morgan Kaufmann.

[14] Pier Luca Lanzi. The xcs library. `http://xcslib.sourceforge.net`, 2002.

[15] Pier Luca Lanzi and Alessandro Perrucci. Extending the Representation of Classifier Conditions Part II: From Messy Coding to S-Expressions. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 99)*, pages 345–352, Orlando (FL), July 1999. Morgan Kaufmann.

[16] Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors. *Learning Classifier Systems: From Foundations to Applications*, volume 1813 of *Lecture notes in Computer Science*. Springer-Verlag, April 2000.

[17] Tim Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[18] Dale Schuurmans and Jonathan Schaeffer. Representational Difficulties with Classifier Systems. In J. David Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms (ICGA89)*, pages 328–333, George Mason University, June 1989. Morgan Kaufmann. http://www.cs.ualberta.ca/jonathan/Papers/Papers/classifier.ps.

[19] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning – An Introduction*. MIT Press, 1998.

[20] Stewart W. Wilson. Classifier System mapping of real vectors. In *Collected Abstracts for the First International Workshop on Learning Classifier System (IWLCS-92)*, 1992. October 6–8, NASA Johnson Space Center, Houston, Texas.

[21] Stewart W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995. http://prediction-dynamics.com/.

[22] Stewart W. Wilson. Get Real! XCS with Continuous-Valued Inputs. In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Learning Classifier Systems. From Foundations to Applications*, volume 1813 of *LNAI*, pages 209–219, Berlin, 2000. Springer-Verlag.

[23] Stewart W. Wilson. Mining oblique data with xcs. In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *IWLCS*, volume 1996 of *Lecture Notes in Computer Science*, pages 158–176. Springer, 2001.